DOI: 10.3745/KIPSTA.2009.16-A.2.55

# 병행 Java 프로그램의 확장적 경합탐지를 위한 JDI 기반의 투명한 감시도구

김 영 주[†] · 구 인 본[††] · 배 병 진[†††] · 전 용 기[††††]


## 요 약

병행 Java 프로그램의 경합은 프로그램의 비결정성을 초래하므로 반드시 탐지되어야 한다. 이러한 경합을 수행 중에 탐지하기 위해서는 스레드에 대한 수행양상과 모든 접근사건들을 감시할 수 있어야 한다. 기존의 경합탐지 기법들은 프로그램의 수행중에 기록된 파일들을 분석하거나 대상 프로그램을 수정하여 감시하므로 스레드나 모든 접근사건들에 대한 감시가 현실적으로 어렵다. 본 연구에서는 JDI(Java Debug Interface)를 이용하여 스레드에 대한 수행양상과 모든 접근사건을 감시하여 확장적 경합탐지를 할 수 있는 투명한 감시도구를 제안한다. 여기서 JDI는 JDPA(Java Platform Debugger Architecture)에서 제공하는 상위 레벨의 100% 순수 자바 인터페이스로써 자바프로그램의 수행중에 특정 정보를 제공할 수 있다. 그리고 제안된 도구의 투명성을 입증하기 위해서 벤치마크 프로그램으로 실험한 결과, 모든 스레드와 접근사건들을 프로그램 수정없이 감시할 수 있었고 프로그램의 감시시간이 20배 이상 증가되었다.


**키워드 : Java 프로그램, JDI, 경합탐지, 확장성, 투명한 감시도구**

# A Transparent Monitor Based on JDI for Scalable Race Detection of Concurrent Java Programs

Young-Joo Kim[†] · In-Bon Kuh[††] · Byoung-Jin Bae[†††] · Yong-Kee Jun[††††]


## ABSTRACT

Race conditions in current Java programs must be detected because it may cause unexpected result by non-deterministic executions. For detecting such races during program execution, execution flows of all threads and all access events can be monitored. It is difficult for previous race detection techniques to monitor all threads and access events in actuality because these techniques analyze the files traced during program execution or modify original source programs and then monitor these programs. This paper presents a transparent scalable monitoring tool to detect races using JDI(Java Debug Interface) where JDI is 100% pure java interface to provide in JDPA(Java Platform Debugger Architecture) and is able to provide information corresponding to events occurred in run-time of programs. This tool thus can monitor execution flows of all threads and all access events without program modification. We prove transparency of the presented tool and grasp the efficiency of it using a set of published benchmark programs. As a result of this, the suggested tool can monitor all threads and accesses of these programs without their modification, and their monitoring time is increased to more than 20 times.


**Keywords : Java Programs, JDI, Race Detection, Scalability, Transparent Monitor**

## 1. 서 론

최근 일반 사용자에게까지 멀티 코어프로세서가 대중화됨으로 인해 병행 프로그래밍이 가능한 Java 프로그램[4, 11]에 대한 관심이 높아지고 있다. 이러한 프로그램에서 발생할 수 있는 심각한 오류 중에서 두 개 이상의 스레드 또는 프로세스가 적절한 동기화 없이 동시에 공유변수에 접근하며 그 중에서 하나 이상이 쓰기사건일 때 이를 경합[7]이라고 한다. 이러한 경합은 비결정적인 수행결과를 초래하므로 반드시 탐지되어야 한다. 이러한 경합의 탐지 시에 사용되는 전통적인 방법은 breakpoint 이다. 이 방법은 프로그램의 실행시간이나 스레드의 실행순서를 변경할 수 있기 때문에 병행 프로그램에서는 비효율적이다.

병행 프로그램에서 경합을 탐지하는 기법은 정적 기법과 동적 기법이 있다. 이들 기법 중에서 현실적인 기법은 동적

† 정 회 원 : 한국과학기술원 연구교수
†† 준 회 원 : 국립경상대학교 정보과학과 석박사통합과정
††† 정 회 원 : 한국기계연구원 선임연구원
†††† 종신회원 : 경상대학교 정보과학과 전임교수
논문접수 : 2008년 11월 22일
수 정 일 : 1차 2009년 2월 12일
심사완료 : 2009년 2월 12일

경합탐지 기법으로 알려져 있다. 이 기법은 사후추적 기법과 수행중 경합탐지 기법으로 분류된다. 사후추적 기법은 프로그램 수행 중에 기록된 파일을 분석하여 경합을 탐지하며, Intel사의 Thread Checker [9]와 Sun사의 Thread Analyzer [15]에서 적용되고 있고, 수행중 경합탐지 기법은 프로그램의 수행과 분석을 동시에 진행하여 경합을 탐지하며, Eraser [12]와 RecPlay [10] 등에서 적용되고 있다. 그러나 이들 기법은 원시코드를 감시 가능한 코드로 변형하여 경합탐지 여부를 검사한다.

이들 기법의 성능을 향상시키기 위해서 프로그램에서 발생하는 공유변수에 대한 모든 접근사건들 중에서 경합의 가능성이 있는 접근사건들만을 감시하는 확장적 경합탐지 기법[1, 5, 8, 16, 17]도 있다. 그러나 이러한 기법들은 대상 프로그램의 원시코드에 스레드나 접근사건을 감시하는 코드를 삽입하여 경합을 탐지하므로 Java에서 제공하는 패키지나 사용자가 작성한 바이트 코드에서 발생하는 스레드 및 접근사건들에 대해서 감시하는 것은 현실적으로 어렵다. 이러한 것을 해결하기 위해서 JDI, JVMTI, BCI 등을 이용할 수 있다.

본 논문에서는 수행 중에 발생하는 모든 접근사건을 감시하기 위해서 JDI (Java Debug Interface)[14]를 이용하고 확장적으로 경합을 탐지하기 위해서 state-of-art기법[8]을 이용하는 투명한 감시도구를 제안한다. JDI는 대상 프로그램을 수정하지 않고 수행 중에 수행상태를 감시할 수 있는 Java API를 제공한다. 그리고 공인된 벤치마크 프로그램을 이용하여 접근사건이 감시되는 횟수와 선택되는 횟수를 분석하여 본 도구의 투명성을 보인다.

2절에서는 JDI(Java Debugging Interface)와 확장적 감시 기법에 대해서 설명하고, 3절에서는 JDI를 이용하여 경합탐지의 확장성이 보장되는 투명한 감시도구에 대해서 설명한다. 그리고 4절에서는 공인된 벤치마크 실험을 통해서 본 도구의 투명성을 입증한다. 마지막으로 5절에서는 결론 및 향후 과제를 제시한다.

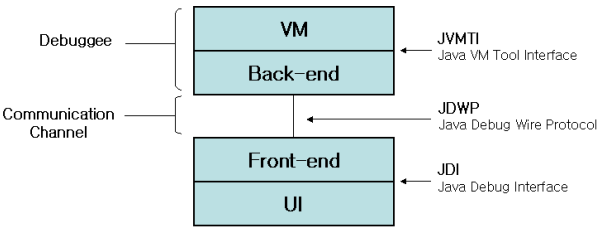
## 2. 연구 배경

본 절에서는 수행 중에 프로그램의 상태를 감시할 수 있는 JDI에 대해서 설명하고, JDI를 이용한 확장적 경합탐지 감시 기법에 대해서 설명한다.

### 2.1 JDI (Java Debug Interface)

JDI[14]는 JPDA(Java Platform Debugger Architecture)[13]의 일부로서 디버깅 프로그램 작성을 지원하는 Java API이다. JPDA는 Java에서 제공하는 디버그 플랫폼이며, 그 구조는 (그림 1)과 같이 Debuggee 측 Back-end인 JVMTI (Java VM Tool Interface)와 Debug UI측 Front-end인 JDI, 그리고 이들 간의 통신규격인 JDWP(Java Debug Wire Protocol)로 이루어져 있다.

JDWP는 디버거와 디버거로 디버그 하는 Java 가상머신과의 통신에 사용되는 프로토콜이다. JDWP는 패킷 기반으로 비동기식 통신을 하며, 패킷은 명령 패킷과 응답 패킷으로 구성되어 있다. 그리고 JVMTI는 JDK 1.4까지는 JVMPI (JVM Profile Interface)로 제공되었고, JDK 1.5부터 Java API들을 통합하여 JVMTI로 제공되기 시작했다. JVMTI는 JVMPI에서 사용하는 이벤트 기반이 아니라, Bytecode Instrumentation 방법을 사용한다. 여기서 Bytecode Instrumentation은 프로그램을 감시하거나 추적하기 위해서 프로그램의 Bytecode를 변경한다는 것이다. 따라서 JVMTI는 자바 가상머신에서 수행하는 프로그램의 수행 상태를 감시하거나 제어할 수 있다. JDI는 수행 중인 Java 프로그램의 상태를 감시할 수 있는 API들을 제공하며 플랫폼에 독립적으로 디버깅할 수 있는 상위레벨 디버깅 환경을 제공한다. Connector API들을 이용하여 로컬 또는 원격지의 Java Program들을 연결할 수 있고 새로운 Java 프로그램을 구동하여 감시하거나 이미 수행 중인 것도 감시할 수 있다. 연결이 완료되면 VirtualMachineManager API를 이용하여 VM에 접근하고 Request API로 Event를 요청하여 원하는 Event 정보를 수집할 수 있다. 따라서 JDI는 프로그램의 수행 중에 발생하는 스레드나 접근사건들에 대한 감시 및 제어를 할 수 있는 환경을 제공한다.




(그림 1) JPDA 구조도

### 2.2 확장적 경합탐지

Java와 같은 병행성을 지원하는 프로그램에서 발생할 수 있는 오류인 경합을 수행 중에 탐지하는 기법들[2, 6, 10, 12]은 공유변수에 대한 매 접근사건을 검사하여 공유자료구조인 접근역사(Access History) 내에 유지되는 이전의 접근사건들과 비교하여 경합을 탐지한다. 이 기법들은 최대 병렬성이 증가할수록 공유자료구조에 심각한 병목현상이 발생할 수 있으므로 경합탐지의 성능이 저하될 수 있다. 이러한 문제점을 해결하기 위해서 확장적 경합탐지 기법들[1, 5, 8, 16, 17]이 연구되어져 왔다.

이 확장적 경합탐지 기법의 특징은 공유자료구조에서 발생하는 병목현상을 줄이기 위해서 경합의 가능성이 있는 접근사건들만을 선택하여 공유자료구조의 접근을 최소화하고 경합탐지를 위한 확장성을 높인다. 그러나 이 기법들[1, 5, 8, 16, 17]은 접근사건을 감시하기 위해서 원시프로그램 또는 중간코드를 수정하므로, 대상 프로그램이 포함시키는 기본 Java 패키지나 외부 Java 패키지는 수정하기가 어렵고, 소스가 공개되지 않는 바이트 코드는 전혀 감시할 수가 없다. 그러므로 수행 중에 발생하는 모든 접근사건을 감시하기가 현실적으로 어렵고, 이를 해결하기 위한 투명한 감시

도구가 필요하다.

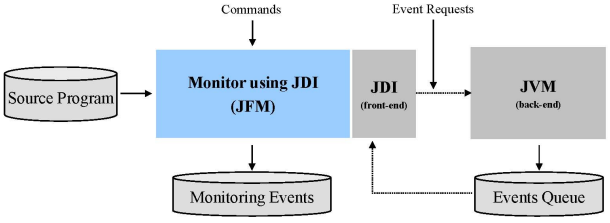
## 3. 투명한 감시도구

본 절에서는 공유변수에 대한 접근사건을 선택하는 투명한 감시도구인 TFM(Transparent Filtering Monitor)을 제안한다. 제안된 도구의 수행환경을 소개하고, 대상 프로그램을 수행시키고 접근사건을 감시하여 선택하는 수행과정을 설명한다.

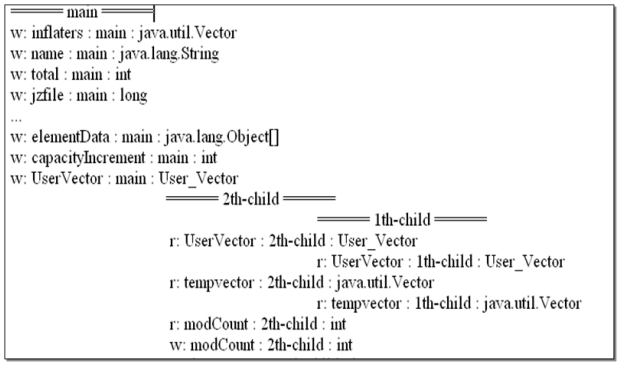
### 3.1 TFM의 수행환경

(그림 2)는 확장적 경합탐지를 할 수 있는 투명한 감시도구인 TFM의 수행환경을 나타낸 것이다. 이 환경은 경합을 탐지하고자 하는 대상 프로그램 정보가 기술되어 있는 Command로부터 시작된다. 대상 프로그램은 자바 프로그램이며, Command 형식은 "java TFM [-option] class file"이다. 이 명령어가 실행되면 TFM은 Java VM (Virtual Machine)을 구동하는 동시에 대상 프로그램에서 발생하는 이벤트들을 감시하기 위한 Java VM제어를 획득하고, 접근사건을 감시하기 위해서 필요한 이벤트들을 Java VM에 요청한다. 요청된 이벤트들은 수행 중에 감시되어 TFM으로 전달되고 분석된 후 경합의 가능성이 있는 접근사건들만 선택되어 Filtered Events에 기록된다.

(그림 2)에서 JFM은 JVM 상에서 수행되는 프로그램에서 발생하는 특정 이벤트를 감시하거나 제어하기 위해서 JFM에 이벤트 핸들을 등록한다. 그 이벤트 핸들은 가상머신의 시작과 정지, 스레드의 시작과 정지, 메소드의 진입과 나감, 공유변수에 대한 읽기와 쓰기접근에 대한 이벤트들로 구성되어 있다. 이들 이벤트들은 JVM에서 프로그램의 수행시에 발생하면 "Events Queue"로 보내진다. 보내진 이벤트들은 TFM에서 "Monitoring Events"에 수집한다. (그림 3)은 간단한 예제 프로그램으로 수행한 결과를 보인 것이다. 이 그림에서 3개의 스레드가 감시되었고, 클래스와 변수들에 대해서 읽기(r)와 쓰기(w) 접근사건들이 감시되었다는 것을




(그림 2) TFM (Transparent Filtering Monitor)의 수행환경




(그림 3) TFM으로 간단한 java 프로그램 실행결과

```
public class TFM { ...
public static void main(String[] args) { ...
private final VirtualMachine vm;
    ...
    VM_Generator VG = new VM_Generator(args);
    vm = VG.requestVm();
    VG.setDebugTraceMode(debugTraceMode);
    Event_Monitor EM = new Event_Monitor(vm);
    EM.setEventRequests();
    Event_Filter EF = new Event_Filter();
    EM.start();
    ...
    EM.join();
}} //End TFM
```

(그림 4) TFM의 주요 수행코드

알 수 있다.

(그림 4)은 TFM 코드 일부를 나타낸 것이다. TFM은 Command에 기술된 대상 프로그램 정보를 이용하여 *VM_Generator*를 생성한 후에 VM 객체를 생성한다. 생성된 VM으로 *Event_Monitor*인 EM 객체를 생성한다. 그리고 EM을 이용하여 Event 요청을 설정한다. 이와 같이 대상 프로그램을 감시할 준비가 끝나면 *Event_Filter*를 생성시키고 감시를 시작시킨다.

### 3.2 TFM의 수행과정

TFM의 내부 모듈은 (그림 5)와 같이 VM Generator, Event Monitor, 그리고 Event Filter로 나누어진다. VM Generator는 Command를 분석하여 대상 프로그램이 수행될 VM을 생성하고 다음 단계의 Event Monitor로 제어를 전달한다.


(그림 5) TFM의 수행과정

Command에 기록된 정보에는 대상 프로그램의 이름, 로그 파일 이름, 멤버변수 이름, 그리고 옵션 등이 포함된다. Event Monitor는 전달 받은 VM을 시작시키고 감시할 공유변수들을 등록하고 그 후에 감시된 Event들을 Event Filter로 보내는 역할을 한다. 그리고 감시할 공유변수 등록을 위해서 수행 중 적재되는 모든 Java 객체의 멤버변수 정보를 수집하여 일괄적으로 멤버변수 접근감시를 요청한다. 감시된 Event에는 접근한 스레드 정보, 읽기·쓰기 사건 유무, Event가 발생된 위치의 source 파일 이름, 줄 번호 등의 경합 탐지에 필요한 정보들이 포함되어 있다. Event Filter는 전달 받은 Event들 중에서 기존의 접근사건 선택기법 [OcCh03]을 이용하여 경합의 가능성이 있는 Event들만 선택하여 Filtered Event에 기록하는 역할을 담당한다.

프로그램의 수행 중에 발생하는 접근사건 감시를 위해서는 먼저 스레드에 대한 시작과 종료시점을 알아야 하므로 JDI API 중에서 ThreadStartEvent와 ThreadDeathEvent 클래스를 이용하고, 다음으로 그 스레드에서 발생하는 공유변수들에 대한 읽기와 쓰기 접근사건들을 감시하기 위해서는 AccessWatchpointEvent와 ModificationWatchpointEvent 클래스를 이용한다. 이들 접근사건들의 선택기준은 각 스레드에서 블록마다 기껏해야 하나의 읽기와 쓰기 접근사건들을 선택하는 것을 기본으로 하고 있다. 여기서 블록은 동기화 명령어나 스레드 관련 명령어에 의해서 구분된 영역을 의미한다.

## 4. 실 험

본 절에서는 실험을 수행한 실험환경 및 벤치마크 프로그램에 대해서 소개하고 이를 이용하여 투명성과 효율성을 측정한 결과를 분석한다.

### 4.1 실험환경

운영체제는 Windows XP를 설치하고 Java Compiler와 Java Runtime Environment는 각각 J2SDK 1.4.1과 JRE 1.4.1를 설치하였다. TFM을 작성하는데 사용된 JDI API 버전은 1.4이다. 실험에서 이용한 벤치마크 프로그램은 병행 Java 프로그램의 실험에 널리 이용되는 The Java Grande Forum Multi-threaded Benchmarks Suite[Epcc07]이다. 이는 세 종류의 벤치마크 프로그램을 제공하는데 본 실험에서 사용한 것은 *Kernel* 프로그램들이다. 이 프로그램들은 Series, LUFact, SOR, Crypt, Sparse 등으로 모두 병행 스레드를 이용하여 수행시간의 대부분을 계속되는 수식연산에 소모하기 때문에 공유변수의 접근이 빈번하고 다량으로 발생하므로 본 연구의 실험에 적합하다.

### 4.2 투명성 실험

<표 1>는 투명성 실험결과이다. *Whole Program*은 JDI를 이용하여 수행 중 발생하는 모든 접근사건을 감시한 항목이고, *Target Program Only*는 JDI를 이용하지만 기본 Java 패키지의 접근사건을 감시하지 않고 실험한 항목이다.

표에서 *#Threads*, *#Shared Variables*, *#Total Accesses*, *#Totally-Filtered Accesses*는 대상 프로그램이 수행하는 스레드 수, 감시된 공유변수의 수, 감시된 전체 접근사건의 수, 그리고 선택된 접근사건의 수를 나타낸 것이다. *static*과 *non static*은 클래스의 정적 변수와 객체의 인스턴스 변수를 나타낸 것이다. 여기서 정적 변수는 해당 클래스를 참조하는 모든 스레드에서 접근 가능하여 공유할 수 있기 때문에 감시되어야 하며, 인스턴스 변수도 스레드마다 개별적으로 메모리에 할당되지만 참조가 다른 스레드로 전달될 경우 공유될 수 있기 때문에 감시대상에 포함되어야 한다.

Series 프로그램을 감시한 결과 감시된 공유변수의 수에서 두 배 이상 차이가 나는 것을 볼 수 있으며 선택된 접근사건에서 세 배 이상 선택된 것을 볼 수 있다. 감시된 전체 접근사건의 수 또한 감시횟수가 많다는 것을 알 수 있다. 다섯 가지 실험에서 모두 Whole Program의 실험결과에서 더 많은 감시사건과 선택이 보고되었다. 이것은 기본 Java 패키지에서도 공유변수 접근사건이 발생함을 보여주고 있고 반드시 감시되어야 함을 보여준다.

### 4.3 효율성 실험

<표 2>은 대상 프로그램의 단독 수행시간과 TFM을 이용했을 때의 수행시간을 보인 것이다. *No TFM*은 TFM을 이용하지 않고 대상 프로그램만을 수행한 것이고, *VM Generator* 는 TFM 에서 접근사건을 감시하지 않고 대상 프로그램을 수행시켰을 때의 수행시간을 나타낸 것이다. 그리고 TFM 은 TFM에서 제공하는 모든 기능을 이용하여

〈표 1〉 Transparency: Benchmark programs(start-only monitoring)

| Fact / Example | Input Size | #Threads | Whole Program | | | | | | Target Program Only | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | #Shared Variables | | #Total Accesses | | #Totally-FilteredAccesses | | #Shared Variables | | #Total Accesses | | #Totally-FilteredAccesses | |
| | | | static | non static | static | non static | static | non static | static | non static | static | non static | static | non static |
| Series | 1000 | 4 | 8 | 26 | 40097 | 173 | 43 | 29 | 4 | 10 | 40072 | 52 | 13 | 13 |
| LUFact | 500 | 4 | 6 | 41 | 4042 | 538694 | 9 | 59 | 2 | 25 | 4017 | 538536 | 5 | 43 |
| SOR | 1000 | 4 | 8 | 78 | 3984896 | 416581 | 12 | 99 | 4 | 14 | 3984871 | 410868 | 8 | 29 |
| Crypt | 1000 | 5 | 6 | 84 | 65 | 33523 | 10 | 113 | 2 | 20 | 40 | 27779 | 6 | 36 |
| Sparse | 10000 | 4 | 11 | 83 | 422106 | 1245021 | 15 | 120 | 7 | 19 | 420060 | 1235175 | 10 | 35 |

〈표 2〉 Efficiency: Benchmark programs(start-only monitoring)

| Example \ Fact | Instrument Type | #Shared Variables | No TFM(ms) | VM Generator(ms) | TFM(ms) |
|---|---|---|---|---|---|
| Series | Source | 14 | 25765 | 27703 | 601000 |
| | JDI | 34 | 25765 | 28203 | 694359 |
| LUFact | Source | 27 | 25765 | 1109 | 13761891 |
| | JDI | 46 | 25765 | 1156 | 13870391 |
| SOR | Source | 18 | 39610 | 16406 | 40963594 |
| | JDI | 18 | 39610 | 17031 | 39740062 |
| Crypt | Source | 22 | 250 | 437 | 470547 |
| | JDI | 42 | 250 | 625 | 478547 |
| Sparse | Source | 26 | 15 | 468 | 42257531 |
| | JDI | 93 | 15 | 593 | 42777985 |

대상 프로그램을 수행시켰을 때의 수행시간을 나타낸 것이다.

Series 프로그램의 *No-TFM*과 VM Generator의 결과는 500ms의 차이를 보이는데 이는 전체 수행시간에 비해 미미한 차이로 볼 수 있다. 나머지 네 가지의 실험에서도 이를 확인할 수 있다. 그러나 *TFM*의 결과는 *No-TFM*보다 20배 이상의 수행시간이 걸리는 것을 볼 수 있는데 이것은 JDI를 이용하는 감시 기법이 수행속도 개선의 필요성이 있다는 것을 보여준다.

## 5. 결론 및 향후 과제

본 연구는 수행 중에 병행 Java 프로그램의 경합을 탐지하는 기존의 기법들의 문제를 해결하기 위해 JDI를 이용한 투명한 접근사건 감시도구를 제안한다. 그리고 공인된 벤치마크 프로그램을 이용하여 감시된 접근사건들을 분석하여 투명성과 효율성에 대해서 실험하였다.

이 도구는 경합존재의 검증을 위한 기법에 사용되어 실용적으로 디버깅할 수 있는 환경을 제공할 수 있고, 프로그램 특성을 분석하여 다양한 분야에서 사용할 수 있는 인터페이스 역할을 할 수 있다. 그리고 효율성 실험에서 보인 제안한 감시도구의 효율성 문제를 개선하여 성능적으로 효율적인 감시도구를 제안하고자 한다.

## 참 고 문 헌


[1] Choi, J., K. Lee, A. Loginov, R. O'Callahan, V. Sarkar and M. Sridharan, "Efficient And Precise Datarace Detection For Multithreaded Object-oriented Programs," *Conf. on Programming Language Design and Implementation* (PLDI), pp.258-269, ACM Press, 2002.

[2] Dinning, A., and E. Schonberg, "Detecting Access Anomalies in Programs with Critical Sections," *2nd Workshop on Parallel and Distributed Debugging* (WPDD), pp.85-96, ACM, May, 1991.

[3] EPCC, "The Java Grande Forum Multi-threaded Benchmarks," 2007.

[4] D. Holmes, *Java: Concurrency, Synchronisation and Inheritance*, 1998.

[5] Jun, Y., and C. E. McDowell, "Scalable Monitoring Technique for Detecting Races in Parallel Programs," *Proc. of the 5th IEEE Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, IEEE, Lecture Notes in Computer Science, 1800: 340-347, Springer-Verlag, Cancun, Mexico, May, 2000.

[6] Mellor-Crummey, J., "On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism," *Proc. of ACM/IEEE Conf. on Supercomputing*, pp.24-33, ACM, 1991.

[7] Netzer, R. H. B., and B. P. Miller, "What Are Race Conditions? Some Issues and Formalizations," *Letters on Prog. Lang. and Systems*, 1(1): 74-88, ACM, March, 1992.

[8] O'Callahan, R. and J. Choi, "Hybrid Dynamic Data Race Detection," *Proc. of ACM SIGPLAN Symp. on Principle and Practice of Parallel Programming* (PPoPP), San Diego, California, ACM, June, 2003.

[9] Petersen, P., and S. Shah, "OpenMP Support in the Intel Thread Checker," *Int'l Workshop on OpenMP Applications and Tools (WOMPAT)*, pp.1-12, June, 2003.

[10] Michiel, R. and K. Bosschere, "RecPlay: A Fully Integrated Practical Record/Replay System," *ACM Transactions on Computer Systems*, pp.133-152, ACM, May, 1999.

[11] B. Sandn, "Coping with Java Threads," Computer, pp.20-27, IEEE, April, 2004.

[12] Stefan, S., Michael, B., Greg, N., and P. Sobalvarro, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Transactions on Computer Systems*, pp.391-411, ACM, November, 1997.

[13] Sun Microsystems, "Java Platform Debugger Architecture," 2005.

[14] Sun Microsystems, "Java Debug Interface," 2006.

[15] Sun Microsystems Inc., *Sun Studio 12: Thread Analyzer User's Guide*, 2007.

[16] 김영주, 이승렬, 전용기, "임계구역을 가진 공유메모리 병렬프로그램에서 효율적인 경합 탐지를 위한 사건 선택기법," 한국정보과학회 춘계학술발표논문집, 27(1): 630-632, 한국정보과학회, 2000. 4.

[17] 김영주, 박소희, 박미영, 이승렬, 전용기, "록킹을 가진 OpenMP 병렬프로그램의 경합탐지를 위한 확장적 감시기법 및 도구," 경상대학교 전산연구, 15: 7-16, 경상대학교, 2000. 12.

**김 영 주**

e-mail : yjkim73@kaist.ac.kr

1999년 국립경상대학교 컴퓨터과학과(학사)

2001년 국립경상대학교 컴퓨터과학과(공학석사)

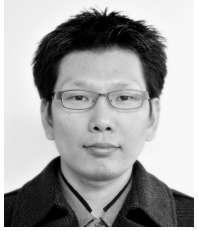
2007년 국립경상대학교 컴퓨터과학과(공학박사)

2007년 9월~2008년 8월 한국정보통신대학교 박사후과정

2008년 9월~2009년 2월 한국정보통신대학교 연구교수

2009년 3월~현 재 한국과학기술원 연구교수

관심분야 : 병렬/분산 컴퓨팅, 임베디드 시스템, 센서 네트워크 (시스템 소프트웨어, 경합탐지, 디버깅, 시각화)



**구 인 본**

e-mail : inbon@gnu.ac.kr

2005년 8월 국립경상대학교 컴퓨터과학과(학사)

2006년 3월~현 재 국립경상대학교 정보과학과 석박사통합과정

관심분야 : 분산병렬처리, 운영체제커널, 커널디버깅

**배 병 진**

e-mail : bjbae@kimm.re.kr

2002년 국립경상대학교 컴퓨터과학과(학사)

2004년 국립경상대학교 컴퓨터과학과(공학석사)

2004년 9월~2006년 7월 한국기계연구원 위촉연구원

2006년 8월~2008년 12월 한국기계연구원 연구원

2009년 1월~현 재 한국기계연구원 선임연구원

관심분야 : 데이터베이스, 임베디드 시스템

**전 용 기**

e-mail : jun@gnu.ac.kr

1980년 경북대학교 컴퓨터공학과(학사)

1982년 서울대학교 컴퓨터과학과(공학석사)

1993년 서울대학교 컴퓨터과학과(공학박사)

1995년 3월~1996년 8월 Univ. of California, Santa Cruz 연구원

1982년 2월~1985년 3월 정보통신연구원(ETRI) 연구원

1985년 3월~현 재 경상대학교 정보과학과 전임교수

관심분야 : 분산병렬처리, 내장형시스템, 시스템소프트웨어